Component Substitution through Dynamic Reconfigurations

Arnaud Lanoix, Olga Kouchnarenko

Component Substitution through Dynamic Reconfigurations

Arnaud Lanoix
LINA CNRS and Nantes University
Nantes, France
arnaud.lanoix@univ-nantes.fr

Olga Kouchnarenko
FEMTO-ST CNRS and University of Franche-Comté
Besançon, France
Inria/Cassis France
olga.kouchnarenko@univ-fcomte.fr

Component substitution has numerous practical applications and constitutes an active research topic. This paper proposes to enrich an existing component-based framework—a model with dynamic reconfigurations making the system evolve—with a new reconfiguration operation which "substitutes" components by other components, and to study its impact on sequences of dynamic reconfigurations. Firstly, we define *substitutability constraints* which ensure the component encapsulation while performing reconfigurations by component substitutions. Then, we integrate them into a *substitutability-based simulation* to take these substituting reconfigurations into account on sequences of dynamic reconfigurations. Thirdly, as this new relation being in general undecidable for infinite-state systems, we propose a semi-algorithm to check it on the fly. Finally, we report on experimentations using the B tools to show the feasibility of the developed approach, and to illustrate the paper's proposals on an example of the HTTP server.

1 Introduction

Dynamic reconfigurations [2, 3, 21] increase the availability and the reliability of component-based systems by allowing their architecture to evolve at runtime. In this paper, in addition to dynamic evolution reconfigurations, possibly guided by temporal patterns [12, 20, 11], we consider reconfigurations bringing into play by component substitutions. These reconfigurations by substitution may change the model's behaviour. The questions we are interested in are: How are such model transformations represented? What aspects of the model's behaviour can be changed? Can new behaviour be added, can existing behaviours be replaced or combined with new behaviours?

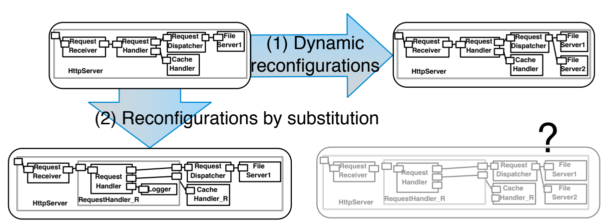

Figure 1: Different kinds of reconfigurations

More precisely, in our previous works [12, 20, 11], a component-based framework has been developed: an component architecture with dynamic reconfigurations has been defined and shown consistent, a linear temporal pattern logic allowing expressing properties over sequences of dynamic reconfigurations has been defined. Component substitution reconfigurations being motivated by numerous practical applications, this paper proposes to enrich the existing component-based framework with a notion of *component substitutability*. Since the model is formulated as a theory in FOL, this is achieved by introducing a new relation over components, and a set of logical constraints. Then, the paper presents a notion of simulation between dynamic reconfigurable systems wrt. a given component substitution relation, and addresses the checking of this relation, which is known to be, in general, undecidable.

Figure 1 displays two kinds of considered reconfigurations: Horizontal reconfigurations represent the dynamic architecture's evolution whereas vertical substitutions lead to different implementations. As

Submitted to:
FESCA @ ETAPS 2014

the model and its implementations must remain consistent through evolution, we study the impact of reconfigurations by substitution on sequences of dynamic reconfigurations.

The main purpose of the present paper consists in studying the impact of reconfigurations by substitution on sequences of dynamic reconfigurations. To this end, we propose to extend the previous work in [20] on the verification of the architectural consistency using the B tools by, first, evaluating component substitutability and, second, by evaluating sequences of reconfigurations—and, consequently, the substitutability-based simulation. For this we propose to use the $\mathbb{B}_4 = \{\perp, \perp^p, \top^p, \top\}$ truth domain which is suitable to evaluate the substitution relation on the fly. Using $\mathbb{B}_4$ is in line with the work in [11] on the runtime verification of linear temporal logic properties.

Layout of the paper. In Sect. 2 we recall the main features of the architectural reconfiguration model introduced in [12, 20] and illustrate them on an example of the HTTP server. In Sect. 3, a new reconfiguration operation by component substitution is introduced and substitutability constraints are defined to ensure component encapsulation. In Sect. 4 component substitutability is integrated into a substitutability-based simulation relation. This relation being undecidable in general, a semi-algorithm is proposed to evaluate on the fly dynamic reconfiguration sequences and, consequently, the component substitutability-based simulation. Section 5 explains how to use the B tools for dealing with component substitutability through dynamic reconfigurations, and describes experiments on the HTTP server example. Finally, we conclude in Sect. 6.

2 Background: Architectural Reconfiguration Model

The reconfigurations we consider here make the component-based architecture evolve dynamically. They are combinations of *primitive* reconfiguration operations such as instantiation/destruction of components; addition/removal of components; binding/unbinding of component interfaces; starting/stopping components; setting parameter values of components.

In general, system configuration is the specific definition of the elements that define or prescribe what a system is composed of. We define a configuration to be a set of architectural elements (components, required or provided interfaces and parameters) together with relations to structure and to link them, as depicted in Fig. 2 [1].

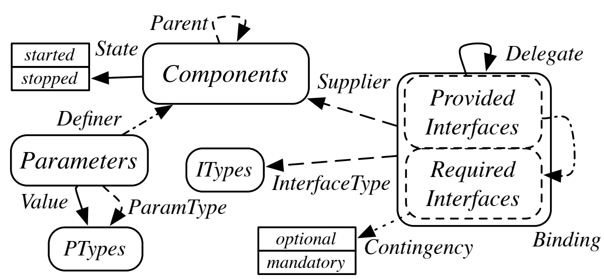

Figure 2: Configurations = architectural elements and relations

Given a set of configurations $\mathscr{C} = \{c, c_1, c_2, \ldots\}$, we introduce a set CP of configuration properties on the architectural elements and the relations between them. These properties are specified using first-order logic formulas. The interpretation of functions, relations, and predicates is done according to basic definitions in [17] and in [11][1]. We now define a configuration *interpretation* function $l : \mathscr{C} \to CP$ which gives the largest conjunction of $cp \in CP$ evaluated to true on $c \in \mathscr{C}$ [2].

Among all the configuration properties, we consider the architectural *consistency constraints CC* which express requirements on component assembly common to all the component architectures. They allow defining *consistent configurations* which notably respect the following rules. Their intuition is as

[1] See Definition 5 in Appendix A

[2] By definition in [17], this conjunction is in CP.

follows, together with a formal description for several constraints[3]:

- a component *supplies* one provided interface, at least;
- the composite components have no parameter;
- a sub-component must not be a composite including its own parent component;
- two bound interfaces must have the same interface type; they are not supplied by the same component, but their containers are sub-components of the same composite;

$$\begin{array}{l}\forall ip \in ProvInterfaces,\\ \forall ir \in ReqInterfaces\end{array} . \left(Binding(ip) = ir \Rightarrow \begin{array}{l} InterfaceType(ip) = InterfaceType(ir) \\ \wedge\, Container(ip) \neq Container(ir) \\ \wedge\, \exists\, c \in Components. \left(\begin{array}{l} (Container(ip), c) \in Parent \\ \wedge (Container(ir), c) \in Parent \end{array} \right) \end{array} \right)$$

- when binding two interfaces, there is a need to ensure that they have not been involved in a delegation yet; similarly, when establishing a delegation link between two interfaces, the specifier must ensure that they have not been involved in a binding yet;
- a provided (resp. required) interface of a sub-component is delegated to at most one provided (resp. required) interface of its parent component; the interfaces involved in the delegation must have the same interface type;
- a component is *started* only if its mandatory required interfaces are bound or delegated.

Definition 1 (Consistent configuration) *Let $c = \langle Elem, Rel \rangle$ be a configuration and CC the architectural consistency constraints. The configuration c is* consistent*, written* `consistent(c)`*, if $l(c) \Rightarrow CC$.*

Let $\mathscr{R}$ be a finite set of reconfiguration operations. The possible evolutions of the component architecture via the reconfiguration operations are defined as a transition system over $\mathscr{R}$.

Definition 2 (Reconfiguration model) *The operational semantics of component systems with reconfigurations is defined by the labelled transition system $S = \langle \mathscr{C}, \mathscr{C}^0, \mathscr{R}, \rightarrow \rangle$ where $\mathscr{C} = \{c, c_1, c_2, \ldots\}$ is a set of* consistent *configurations, $\mathscr{C}^0 \subseteq \mathscr{C}$ is a set of initial configurations, $\mathscr{R}$ is a finite set of reconfigurations, $\rightarrow \subseteq \mathscr{C} \times \mathscr{R} \times \mathscr{C}$ is the reconfiguration relation.*

Let us write $c \stackrel{ope}{\rightarrow} c'$ when a target configuration c' is reached from a configuration c by a reconfiguration operation $ope \in \mathscr{R}$. Given the model $S = \langle \mathscr{C}, \mathscr{C}^0, \mathscr{R}, \rightarrow \rangle$, an evolution path σ (or a path for short) in S is a (possibly infinite) sequence of configurations $c_0, c_1, c_2, \ldots$ such that $\forall i \geq 0 . (\exists\, ope_i \in \mathscr{R}.(c_i \stackrel{ope_i}{\rightarrow} c_{i+1} \in \rightarrow))$. We write $\sigma(i)$ to denote the i-th configuration of a path σ. Let Σ denote the set of paths, and Σ^f ($\subseteq \Sigma$) the set of finite paths.

To illustrate our model, let us consider an example of the HTTP server [4]. The architecture of this server is depicted in Fig. 3. The RequestReceiver component reads HTTP requests from the network and transmits them to the RequestHandler component. In order to keep the response time as short as

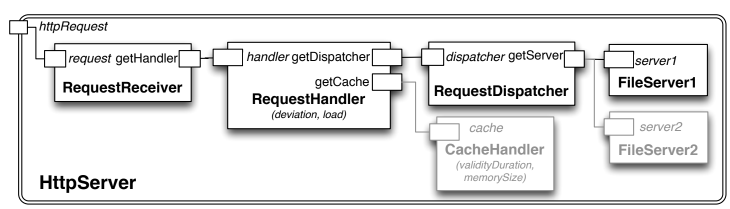

Figure 3: HTTP server architecture

[3] The whole definition is available in Appendix B.
[4] http://fractal.ow2.org/tutorial

possible, **RequestHandler** can either use a cache (with the component **CacheHandler**) or directly transmit the request to the **RequestDispatcher** component. The number of requests (load) and the percentage of similar requests (deviation) are two parameters defined for the **RequestHandler** component. The **CacheHandler** component is used only if the number of similar HTTP requests is high. The memorySize for the **CacheHandler** component depends on the overall load of the server.

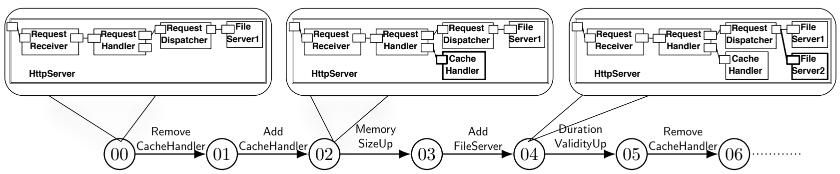

Figure 4: Part of a path of the HTTP server architecture

The validityDuration of data in the cache also depends on the overall load of the server. The number of used file servers (like the **FileServer1** and **FileServer2** components) used by **RequestDispatcher** depends on the overall load of the server. On this example, the considered reconfiguration operations are:

- AddCacheHandler and RemoveCacheHandler which are used to add and remove **CacheHandler**;
- AddFileServer and removeFileServer which are used to add and remove **FileServer2**;
- MemorySizeUp and MemorySizeDown which are used to increase and to decrease the MemorySize value;
- DurationValidityUp and DurationValidityDown which are used to increase and to decrease the ValidityDuration value.

A possible evolution path of the HTTP server architecture is given in Fig. 4.

3 New Reconfigurations by Component Substitution

In this section we enrich our component-based framework with a new kind of reconfigurations allowing a *structural* substitution of the components with respect the component encapsulation. In fact, we want the substituted component to supply the same interfaces of the same types as before. This way the other components do not see the difference between the component and its new "substituted" version, and thus there is no need to adapt them. As the substitution of a component should not cause any changes outside of this component, only the two following kinds of component *substitutions* are allowed:

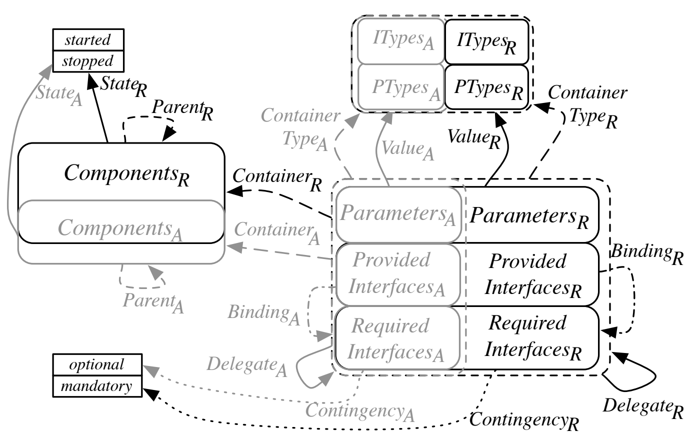

Figure 5: Architectural elements and relations before(grey) and after(black) reconfiguration by substitution

- either a component can be replaced by a new version of itself, or
- a component can be replaced by a composite component which encapsulates new sub-components providing at least the same functionalities as before substitution.

For the allowed substitution cases, Figure 5 displays how the architectural elements and relations are defined at two pre- and post-substitution levels. Let c_A and c_R be two architectural configurations at respectively a pre-substitution and a post-substitution levels. The *substitute reconfiguration* is then expressed by a *partial* function $Subst : Components_A \to Components_R$ that gives how the components are substituted in c_A to obtain c_R.

Let us illustrate our proposal on the example of the HTTP server. For the configuration in Fig. 6, we apply the following substitute reconfiguration:

- **CacheHandler** is replaced by a new version of itself, named **CacheHandler_R**;
- **RequestHandler** becomes a composite component, called **RequestHandler_R**, which encapsulates two new components: **RequestAnalyzer** and **Logger**. **RequestAnalyzer** handles requests to determine the values of the deviation and load parameters. **Logger** allows **RequestAnalyzer** to memorise requests to chose either **RequestDispatcher** or **CacheHandler**, if it is available, to answer requests.

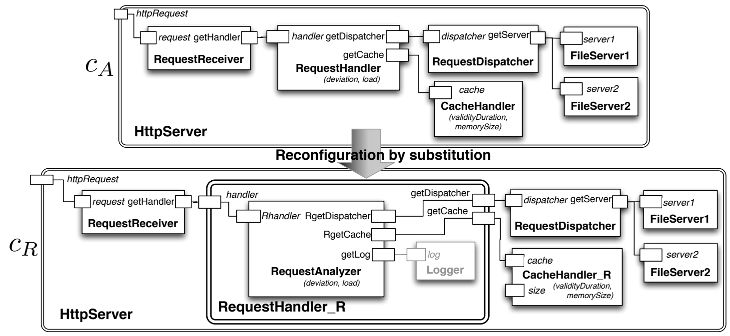

Figure 6: Applying a reconfiguration by substitution on the **HttpServer** example

We have $\left(\begin{array}{l} Subst(\textsf{CacheHandler}) = \textsf{CacheHandler\_R} \\ Subst(\textsf{RequestHandler}) = \textsf{RequestHandler\_R} \end{array}\right)$ as substitute reconfiguration function.

In order to ensure that proposed substitutions respect the requirements on components and their assembly, we now introduce *architectural constraints* on both replaced (or old) and substituted (or new) components. These architectural constraints, named SC_{Subst}, describe which changes are allowed or prescribed by a substitute reconfiguration. Their intuition is as follows, together with a formal description for several constraints[5]:

- *In the system parts not concerned by the component substitution, all the core entities and all the relations between them remain unchanged through the substitution process:*
  - the old parameters and the associated types remain unchanged in the substitutes;
  - the old components remain unchanged;

$$\begin{array}{r} \forall c \in Components_A \cap Components_R, \\ \forall x \in Interfaces_A \uplus Parameters_A \end{array} .(Container_A(x) = c \Rightarrow Container_R(x) = c)$$

  - the old interfaces and their types are not changed;
  - the old connections between component's interfaces are kept as well.

- *For the old components impacted by the components substitution, the constraints are as follows:*
  - an old component completely disappears only if it is substituted by a new version for itself;

$$\forall c_A. \left(\begin{array}{r} c_A \in Components_A \\ \setminus Components_R \end{array} \Rightarrow \left(\begin{array}{r} \exists c_R \in Components_R \\ \setminus Components_A \end{array} .(Subst(c_A) = c_R) \right) \right)$$

[5]The whole definition is available in Appendix C.

  - the substituted components are in the same state as the old ones, and either they have the same parent component as before substitution, or the old parent component has been substituted as well;
  - the interfaces of the replaced components are supplied by the substituted components;
  - the parameters of the replaced components are defined either on the substituted components, or on their subcomponents.
- *The new elements introduced during the substitution process cannot impact the old conserved architecture:*
  - the newly introduced components must be subcomponents of some substituted components;

$$\begin{array}{c}\forall c_R \in Components_R \setminus Components_A, \\ \forall c_A \in Components_A \setminus Components_R\end{array} . \left(Subst(c_A) \neq c_R \Rightarrow \begin{array}{c}\exists c'_R \in Components_R \setminus Components_A . \\ ((c_R, c'_R) \in Parent_R)\end{array} \right)$$

  - the newly introduced interfaces must be associated with the new components;

$$\forall i. \left(i \in \begin{array}{c} ProvInterfaces_R \\ \setminus ProvInterfaces_A \end{array} \Rightarrow Container_R(i) \in \begin{array}{c} Components_R \\ \setminus Components_A \end{array} \right)$$

  - the newly introduced parameters are associated with the new components;
  - the new connections are used to connect the new components.

Definition 3 (Structural substitutability) *Let c_A and c_R be two* consistent *configurations, Subst the substitution function, and SC_{Subst} the architectural substitutability constraints. The configuration c_R is* substitutable *to c_A, written* `subst(`c_R`, `c_A`)`*, if $l(c_R) \wedge SC_{Subst} \Rightarrow l(c_A)$.*

4 Component Substitution through Dynamic Evolution

The new reconfigurations by component substitution defined in Sect. 3 must be taken into account in evolutions of component-based architectures. Indeed, as the substituted or the newly introduced components may introduce *new* dynamic reconfigurations, the architectures with substituted components may evolve by the old reconfigurations as well as by new reconfigurations. We want these reconfigurations to be consistent with the reconfigurations by substitution.

To this end, we integrate the architectural substitutability constraints from Sect. 3 into a simulation relation linking dynamic reconfigurations of a system after component's substitutions with their old counterparts that where possible before the component substitution. We then define a substitution relation ρ in the style by Milner-Park [25] as a simulation having the following properties, which are common to other formalisms like action systems [9] or LTL refinement [19]:

1. Adding the new dynamic reconfiguration actions should not introduce deadlocks[6].
2. Moreover, the new dynamic reconfiguration actions should not take control forever: the livelocks formed by these actions are forbidden.

Definition 4 (Substitutability-based simulation) *Let $S_A = \langle \mathscr{C}_A, \mathscr{C}_A^0, \mathscr{R}_A, \rightarrow_A \rangle$ and $S_R = \langle \mathscr{C}_R, \mathscr{C}_R^0, \mathscr{R}_R, \rightarrow_R \rangle$ be two reconfiguration models. Let σ_R be a path of S_R. A relation $\sqsubseteq_{\texttt{subst}} \subseteq \mathscr{C}_R \times \mathscr{C}_A$ is the* substitutability-based simulation *iff whenever $c_R \sqsubseteq_{\texttt{subst}} c_A$ then it implies: structural substitutability* (i)*, strict simulation* (ii)*, stuttering simulation* (iii)*, non introduction of divergence* (iv)*, and non introduction of deadlocks* (v)*, defined as follows:*

[6] We write $c_R \not\rightarrow$ to mean that $\forall ope, c' . \, c \stackrel{ope}{\rightarrow} c' \notin \rightarrow$.

$$\texttt{subst}(c_R, c_A) \quad \text{(i)}$$

$$\forall c'_R \in \mathscr{C}_R, ope \in \mathscr{R}_R \cap \mathscr{R}_A.(c_R \overset{ope}{\rightarrow} c'_R \;\Rightarrow\; \exists c'_A \in \mathscr{C}_A.(c_A \overset{ope}{\rightarrow} c'_A \wedge c'_R \sqsubseteq_{\texttt{subst}} c'_A)) \quad \text{(ii)}$$

$$\forall c'_R \in \mathscr{C}_R, ope' \in \mathscr{R}_R \setminus \mathscr{R}_A.(c_R \overset{ope'}{\rightarrow} c'_R \;\Rightarrow\; c'_R \sqsubseteq_{\texttt{subst}} c_A) \quad \text{(iii)}$$

$$\forall c'_R \in \mathscr{C}_R, ope' \in \mathscr{R}_R \setminus \mathscr{R}_A, k.(k \geq 0 \wedge c_R = \sigma_R(k) \wedge c_R \overset{ope'}{\rightarrow} c'_R \Rightarrow \begin{array}{l} \exists k', ope \in \mathscr{R}_R \cap \mathscr{R}_A. \\ (k' > k \wedge \sigma_R(k') \overset{ope}{\rightarrow} \sigma_R(k'+1))) \end{array} \quad \text{(iv)}$$

$$\forall c_A \in \mathscr{C}_A, \forall c_R \in \mathscr{C}_R.(c_R \sqsubseteq_{\texttt{subst}} c_A \wedge c_R \not\rightarrow \;\Rightarrow\; c_A \not\rightarrow) \quad \text{(v)}$$

We call the substitutability-based simulation (or the substitutability for short) the greatest binary relation over the configurations of S_R and S_A satisfying the above definition. We say that S_R *is simulated by* S_A wrt. the component substitutability, written $S_R \sqsubseteq_{\texttt{subst}} S_A$, if $\forall c_R.(c_R \in \mathscr{C}^0_R \Rightarrow \exists c_A.(c_A \in \mathscr{C}^0_A \wedge c_R \sqsubseteq_{\texttt{subst}} c_A))$.

The substitutability-based simulation defined above can be viewed as a divergence sensitive stability respecting completed simulation in van Glabbeek's spectrum [15]. Since the models are infinite state, the problem to know whether the substitutability-based simulation holds or not is undecidable in general. Consequently, we provide a semi-algorithm to check the substitutability-based simulation on the fly.

The substitutability-based simulation cannot be evaluated to true or false during the system's execution: actually, as the clauses of the substitutability relation $\sqsubseteq_{\texttt{subst}}$ depend not only on the current configurations but also on the target configurations, and even more on sequences of future configurations as in (iv), in general they cannot be evaluated to true or false on the current pair of configurations. But, on the other hand, if one of the clauses of Def. 4 is evaluated to *false* on finite parts of the reconfiguration sequences, then obviously the whole relation does not hold. So, instead of considering the whole transition systems, let us consider a sequence of reconfigurations before substitutions and its counterpart obtained by applying reconfigurations by substitution.

Data: $c^0_R \in \mathscr{C}^0_R$, $c^0_A \in \mathscr{C}^0_A$, $\mathscr{R}_R$ and $\mathscr{R}_A$
Result: $res \in \{\perp, \top\}$, if terminates
$c_R \leftarrow c^0_R$;
$c_A \leftarrow c^0_A$;
while $\top$ **do**
    if `subst(`c_R, c_A`)` **then**
        $\mathscr{E}_R \leftarrow$ `enabled(`c_R, $\mathscr{R}_R$`)` ;
        $\mathscr{E}_A \leftarrow$ `enabled(`c_A, $\mathscr{R}_A$`)` ;
        if $\mathscr{E}_R = \emptyset$ **then**
            if $\mathscr{E}_A = \emptyset$ **then** $res \leftarrow \top$; **break** ;
            else $res \leftarrow \perp$; **break** ;
            end if
        else
            $ope \leftarrow$ `pick-up(`$\mathscr{E}_R$`)` ;
            $c_R \leftarrow$ `apply(`ope, c_R`)` ;
            if $ope \in \mathscr{R}_R \setminus \mathscr{R}_A$ **then** print($\perp^p$) ;
            else
                if $ope \in \mathscr{R}_R \cap \mathscr{R}_A$ **and** $ope \in \mathscr{E}_A$ **then**
                    $c_A \leftarrow$ `apply(`ope, c_A`)` ;
                    print($\top^p$) ;
                else $res \leftarrow \perp$; **break** ;
                end if
            end
        end
    else $res \leftarrow \perp$; **break**;
    end if
end

Figure 7: Semi-algorithm on the substitutability

We propose a semi-algorithm displayed in Fig. 7 to evaluate the substitutability-based simulation starting from the initial configurations $c^0_R \in \mathscr{C}^0_R$, $c^0_A \in \mathscr{C}^0_A$. This semi-algorithm uses the following auxiliary functions:

- `consistent(`$c \in \mathscr{C}$`)` $\in \{\perp, \top\}$ – to determine whether the configuration c is consistent (cf. Def. 1);
- `subst(`$c_R \in \mathscr{C}_R, c_A \in \mathscr{C}_A$`)` $\in \{\perp, \top\}$ – to determine whether the configuration c_R is substitutable to c_A (cf. Def 3) ;

- `enabled(`$c \in \mathscr{C}$`,` $R \subseteq \mathscr{R}$`)` $\subseteq \mathscr{R}$ – to determine the subset of reconfigurations in $\mathscr{R}$ which can be enabled from c;
- `pick-up(`$\mathscr{E} \subseteq \mathscr{R}$`)` $\in \mathscr{R}$ – to choose an operation among reconfigurations in $\mathscr{E}$;
- `apply(`$c \in \mathscr{C}$`,`$ope \in \mathscr{R}$`)` $\in \mathscr{C}$ – to compute the target configuration when applying *ope* to c.

Let us have a close look at the semi-algorithm. When it terminates and returns $\top$ (line 9), finite paths have been considered, no more reconfiguration can be fired at both pre- and post-substitution levels, and all clauses of Def. 4 are satisfied on these finite paths. The semi-algorithm returns $\perp$ in the following three cases.

- Either Line 25 indicates that clause (i) of Def. 4 concerning the structural substitutability from Def. 3 is broken.
- Or Line 10 indicates that there is a deadlock at the level after substitutions but not at the level before components substitutions. In this case clause (v)—the non-introduction of deadlocks—of Def. 4 is broken.
- Or Line 21 indicates that clause (ii)—the strict simulation—of Def. 4 is broken.

Otherwise, we cannot conclude because the semi-algorithm can choose reconfigurations to be applied, and the substitution verification goes on, possibly over infinite paths. Nevertheless, even in this inconclusive case, the semi-algorithm can provide some indications on the current status of the substitutability. Let us consider the set $\mathbb{B}_4 = \{\perp, \perp^p, \top^p, \top\}$ where $\perp, \top$ stand resp. for *false* and *true* values where as $\perp^p, \top^p$ stand resp. for *potential false* and *potential true* values. Like for evaluating temporal properties at runtime as in [11], *potential true* and *potential false* values are chosen whenever an observed behaviour has not yet lead to a violation of the substitutability-based simulation. With this in mind, when a new reconfiguration is applied, $\perp^p$ in Line 15 indicates

- either a potential trouble with the stuttering simulation: clause (iii) of Def. 4 may be broken if, on the next iteration of the semi-algorithm, the structural substitutability—clause (i)—does not hold between the configuration reached on the path with substitutions and the old configuration on the path before component substitutions;
- or a potential divergence: clause (iv) of Def. 4 may be broken if no old reconfiguration occurs in the future.

Finally, when the semi-algorithm indicates $\top^p$, at Line 18, it means that the clauses of Def. 4 have not yet been violated, and the verification of the substitutability-based simulation must continue.

Proposition 1 *Given S_A and S_R, if the substitutability semi-algorithm terminates by providing the $\perp$ value then one has $S_R \not\sqsubseteq_{\texttt{subst}} S_A$.*

The idea behind Proposition 1 is as follows: if there are two sequences of dynamic reconfigurations on which one of the substitutability relation clauses is violated then it does imply the substitutability-based simulation violation.

Let us illustrate the evaluation of the substitutability relation on the example displayed in Fig. 8. As new dynamic reconfigurations introduced by the component substitution, we consider AddLogger and RemoveLogger which consist respectively in adding or removing the newly introduced **Logger** component (see Fig. 6). When a new reconfiguration is executed (leading for example to 14 linked to 02), the evaluation gives $\perp^p$, although the structural substitutability holds. It is due to the fact that the new reconfigurations may take control forever, depending of course on future reconfigurations. In contrast, when an

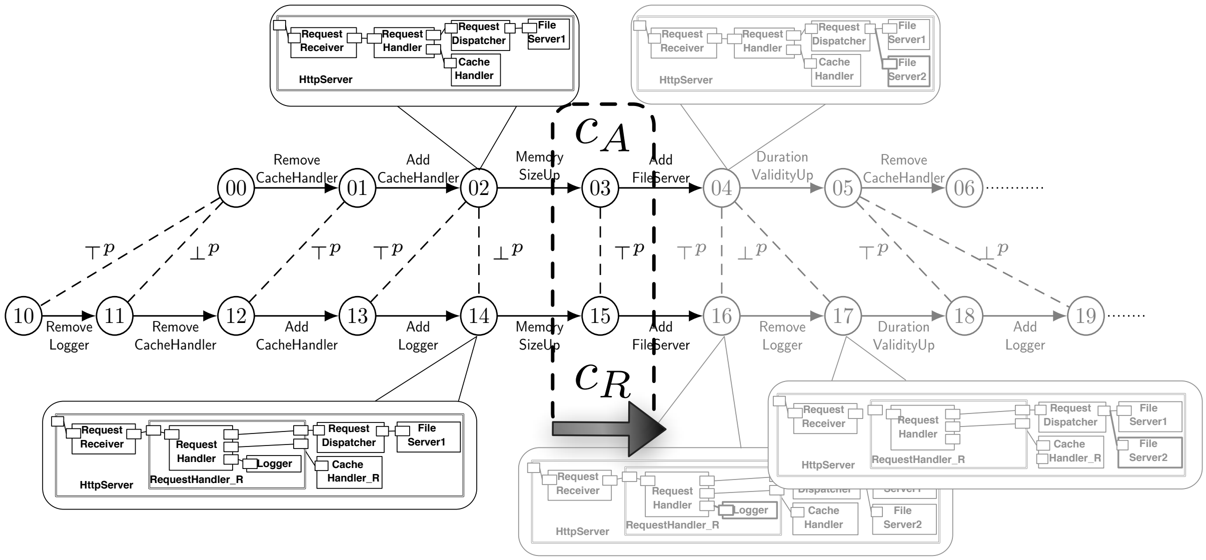

Figure 8: Substitutability evaluation at runtime

old reconfiguration is executed (leading for example to 15 which is linked to 03), the evaluation becomes $\top^p$: the structural substitutability holds and the potential livelock has been avoided. Consequently, when considering finite parts of paths in Fig. 8 until the current pair (c_R, c_A), the reconfigurations of the HTTP server combine well with reconfigurations due to component substitutions.

5 Experiments

This section provides a proof of concept by reporting on experiments using the B tools to express and to check the consistency and substitutability constraints, and to implement the substitutability semi-algorithm.

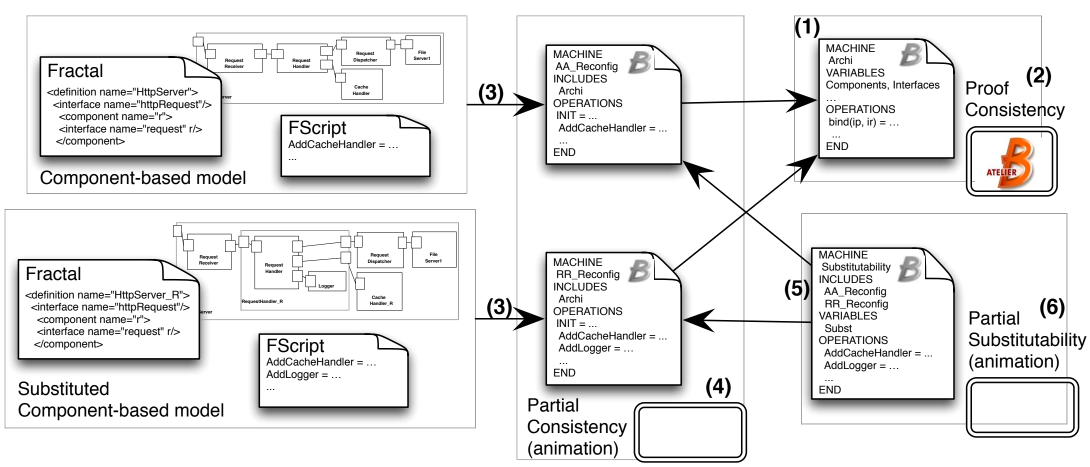

Figure 9: Principle of the validation framework

5.1 A Formal Toolset: the B Method

B is a formal software development method used to model systems and to reason about their development [1]. When building a B machine, the principle is to express system properties—invariants—which are always true after each evolution step of the machine, the evolution being specified by the B operations. The verification of a machine correctness is thus akin to verifying the preservation of these properties, no matter which step of evolution the system takes.

The B method is based on set theory, relations and first-order logic. Constraints are specified in the **INVARIANT** clause of the machine, and its evolution is specified by operations in the **OPERATIONS** clause. Let us assume here that the initialisation is a special kind of operation. In this setting, the *consistency* checking of a B machine consists in verifying that each operation satisfies the **INVARIANT** assuming its precondition and the invariant hold.

Tool supports, such as B4free or AtelierB[7], automatically generate proof obligations (POs) to ensure the consistency in sense of B [1]. Some of them are obvious POs whereas the other POs have to be proved interactively if it was not done fully automatically by the different provers embedded into AtelierB. Another tool, called ProB[8], allows the user to animate B machines for their debugging and testing. On the verification side, ProB contains a constraint-based checker and a LTL bounded model-checker with particular features; Both can be used to validate B machines [22, 23].

5.2 Consistency Checking by Proof and Model Animation

This section summarises the work in [20] on specifications in B of the proposed component-based model with reconfigurations, and on verification process using the B tools, by combining proof and model-checking techniques. Let us consider the B machines which, for readability reasons, are simplified versions of the "real" B machines.

```
MACHINE
 Archi
VARIABLES
 Components, Interfaces, ProvInterfaces, ReqInterfaces, Supplier, Parent, Binding, ...
INVARIANT
   ProvInterfaces ⊆ Interfaces ∧ ReqInterfaces ⊆ Interfaces
 ∧ ProvInterfaces ∪ ReqInterfaces = Interfaces ∧ ProvInterfaces ∩ ReqInterfaces = ∅
 ∧ Supplier ∈ Interfaces → Components
 ∧ Parent ∈ Components ↔ Components
 ∧ Binding ∈ ProvInterfaces ⇸ ReqInterfaces
 ∧ closure1(Parent) ∩ id(Components) = ∅                                  /* CC.3 */
 ∧ ∀ (ip, ir).(ip ↦ ir ∈ Binding ⇒ Provider(ip) ≠ Requirer(ir) ∧ Parent(Supplier(iprov)) = Parent(Supplier(ireq)) ) /* CC.4 + CC.5 */
 ∧ ...
OPERATIONS
bind(ip, ir) =
PRE
   ip ∈ ProvInterfaces ∧ ir ∈ ReqInterfaces ∧ ip↦ir ∉ Binding ∧ ip ∉ dom(Binding) ∧ ip ∉ dom(Delegate) ∧ ir ∉ dom(Delegate)
THEN
   Binding(ip) := ir
END ;
  ...
END
```

The configuration model given in Def. 5 can be easily translated into a B machine Archi ((**1**) in Fig. 9). In this machine, the sets as Components or Interfaces, and relations as Parent or Binding are defined into the **VARIABLES** clause; the architectural consistency constraints *CC* are defined into the **INVARIANT** clause; the basic reconfigurations operations as $bind(ip,ir)$ or $start(compo)$ are also defined here as B operations. Then, we use the AtelierB tool to interactively demonstrate the consistency of the architectural constraints ((**2**) in Fig. 9) through the basic reconfiguration operations.

[7] http://www.b4free.com / http://www.atelierb.eu

[8] http://www.stups.uni-duesseldorf.de/ProB

```
MACHINE
 Reconfig
INCLUDES
 Archi
OPERATIONS
INIT =
 BEGIN
  Components := { HttpServer, RequestReceiver, RequestHandler, CacheHandler, RequestDispatcher, FileServer1, FileServer2 }
  ∥ ProvInterfaces := { httpRequest, request, handler, cache, dispatcher, server1, server2 }
  ∥ ReqInterfaces := { getHandler, getDispatcher, getCache, getServer }
  ∥ Parent := { RequestReceiver↦HttpServer, RequestHandler↦HttpServer, CacheHandler↦HttpServer, RequestDispatcher↦HttpServer }
  ∥ Binding := { handler↦getHandler, cache↦getCache, dispatcher↦getDispatcher, server1↦getServer }
   ...
 END ;
 AddCacheHandler =
 BEGIN
   instantiate (CacheHandler) ;
  add(CacheHandler, HttpServer) ;
  bind(cache, getCache) ;
   start (CacheHandler)
 END ;
 ...
END
```

Then, the generic B machine Archi is instantiated as Reconfig to represent an architecture under consideration, particularly by giving values to all the sets and relations to represent the considered component architecture configuration and by implementing the non-primitive reconfiguration operations using the basic ones ((**3**) in Fig. 9). At this point, we can perform a (partial) validation of the instantiated B machine Reconfig through animations, thanks to the ProB model-checker features ((**4**) in Fig. 9).

5.3 Substitutability Checking by Model Animation

We exploit the work in [20] by considering two instantiated B models AA_Reconfig and RR_Reconfig which define two component architectures, wrt. the pre-/post-substitution levels. All the elements and relations are defined twice: AA.Components, RR.Components, AA.Interfaces, RR.Interfaces, AA.Parent or RR.Parent ... A new machine Substitutability includes these two models ((**5**) in Fig. 9). It defines the *substitute reconfiguration function* Subst to link together the AA.Components to the substituted RR.Components.

```
MACHINE
  Substitutability
INCLUDES
 AA_Reconfig
 RR_Reconfig
VARIABLES
 Subst
INVARIANT
 Subst ∈ AA.Components ⇸ RR.Components
 ∧ ∀(c, i ).( cc ∈ AA.Components ∩ RR.Components ∧ i ∈ AA.Interfaces ∧ AA.Supplier(i) = cc ⇒ RR.Supplier( i ) = c)      /∗ SC.5 ∗/
 ∧ ∀(ca ).( AA.Components − RR.Components ⇒ ∃(cr).(RR.Components − AA.Components ∧ Subst(ca) = cr))           /∗ SC.7 ∗/
 ∧ AA. Interfaces ⊆ RR. Interfaces ∧ AA.ProvInterfaces ⊆ RR.ProvInterfaces ∧ AA.ReqInterfaces ⊆ RR.ReqInterfaces      /∗ SC.13 ∗/
 ∧ ∀( i ).( i ∈ RR.ProvInterfaces − AA.ProvInterfaces ⇒ RR.Supplier( i ) ∈ RR.Components − AA.Components)          /∗ SC.17 ∗/
  ...
INITIALISATION
 Subst := {CacheHandler↦CacheHandlerR, RequestHandler↦RequestHandlerR}
OPERATIONS
 AddCacheHandler =
 BEGIN
  AA_AddCacheHandler ∥ RR_AddCacheHandler
 END;
 AddLogger =
 BEGIN
  RR_AddLogger
 END ;
  ...
END
```

The architectural substitutability constraints SC_{Subst} are defined into the **INVARIANT** clause of this machine; they are constraints between the elements and relations of AA_Reconfig, and the elements and relations of RR_Reconfig. For example, the reader can see some clauses expressed above as a part of the **INVARIANT**.

Afterwards, we use the ProB model-checker to animate the Substitutability machine and to explore—simultaneously— the two instantiated B models, i.e. the pre-/post-substitution component architectures ((**6**) in Fig. 9). This animation allows us to perform the evaluations needed for the semi-algorithm from Section 4: we choose the next dynamic reconfiguration to be applied on the "Enabled operations" windows of ProB (see Fig. 10); if it is an old reconfiguration operation, it is simultaneously executed into AA_Reconfig and RR_Reconfig, otherwise it is only run into RR_Reconfig; then, the **INVARIANT** checking corresponds to the validation of all the SC_{Subst} constraints.

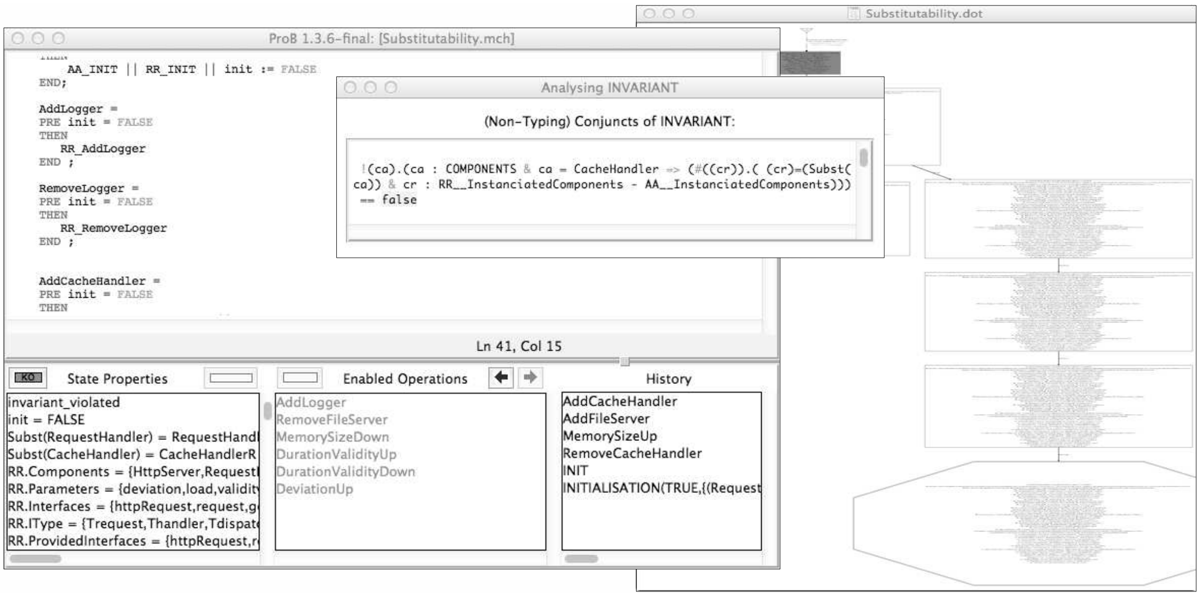

Figure 10: The ProB tool: invariant broken illustrating substitutability constraints broken

Let us suppose that after the reconfiguration by component substitution the AddCacheHandler dynamic reconfiguration contains an implementation error: it does not add the **CacheHandler_R** component. When using ProB, we have easily found the error. Indeed, when AddCacheHandler is executed simultaneously by AA_Reconfig and RR_Reconfig, the invariant is broken as depicted on Fig. 10. More precisely, the clause (**SC.7**) of SC_{Subst} is broken, as the **CacheHandler** component has no substituted component w.r.t. the *Subst* function.

6 Discussion and Conclusion

Related work. For distributed components like Fractal, GCM and ProActive components, the role of automata-based analysis providing a formal basis for automatic tool support is emphasised in [6]. In the context of dynamic reconfigurations, ArchJava [4] gives means to reconfigure Java architectures, and to guarantee communication integrity at run-time. In [5] a temporal logic based framework to deal with systems evolution is proposed.

To compare processes or components, the bisimulation equivalence by Milner [24] and Park [26] is widely used: It preserves branching behaviours and, consequently, most of the dynamic properties; there is a link between the strong bisimulation and modal logics [18]; this is a congruence for a number of composition operators. There are numerous works dealing with component substitutability or interoperability [27, 10, 8]. Our work is close to that in [10], where a component substitutability is defined using equivalences between component-interaction automata wrt. a given set of observable labels. In the present work, in addition to a set of labels, divergency, livelocks are taken into account when comparing execution paths. As KLAPER [16], Palladio [7] and RoboCop [14] component models do not define any refinement/substitution notion, they are clearly distinguishable form our work.

Let us remark that the substitutability-based simulation in this paper is close to the refinement relation in [13]. However, as [13] focuses on a linear temporal logic property preservation, no method is given in [13] to verify the structural refinement.

Conclusion. This paper extends the previous work on the consistency verification of the component-based architectures by introducing a new reconfiguration operation based on components substitutions, and by integrating it into a simulation relation handling dynamic reconfigurations. A semi-algorithm is proposed to evaluate on the fly the substitutability relation and its partial correctness is established. As a proof of concept, the B tools are used for dealing with the substitutability constraints through dynamic reconfigurations. As the ProB tool can deal with a dialect of linear temporal logic, we intend to accompany the present work on component substitutability with a runtime (bounded) model-checking of linear temporal logic patterns. Further, we plan to combine our results with adaptation policies: the partial evaluations $\perp^p$ and $\top^p$ could be informations taken into account into the adaption policies framework to choose the most appropriate reconfiguration that will be applied.

References

[1] J.-R. Abrial (1996): *The B Book - Assigning Programs to Meanings*. Cambridge University Press.

[2] M. Aguilar Cornejo, H. Garavel, R. Mateescu & N. De Palma (2001): *Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications*. Research Report RR-4222, INRIA.

[3] N. Aguirre & T. Maibaum (2002): *A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems*. Automated Software Engineering.

[4] J. Aldric (2008): *Using Types to Enforce Architectural Structure*. In: WICSA'08, pp. 23–34.

[5] H. Barringer, D. M. Gabbay & D. E. Rydeheard (2007): *From Runtime Verification to Evolvable Systems*. In: RV, LNCS 4839, Springer, pp. 97–110.

[6] T. Barros, R. Ameur-Boulifa, A. Cansado, L. Henrio & E. Madelaine (2009): *Behavioural models for distributed Fractal components*. Annales des Télécommunications 64(1-2), pp. 25–43.

[7] S. Becker, H. Koziolek & R. Reussner (2007): *Model-Based performance prediction with the palladio component model*. In: Proceedings of the 6th International Workshop on Software and Performance, WOSP 2007, ACM, pp. 54–65.

[8] P. Brada & L. Valenta (2006): *Practical Verification of Component Substitutability Using Subtype Relation*. In: 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA 2006, IEEE, pp. 38–45.

[9] M. J. Butler (1996): *Stepwise Refinement of Communicating Systems*. Sci. Comput. Program. 27(2), pp. 139–173.

[10] I. Cerná, P. Vareková & B. Zimmerova (2007): *Component Substitutability via Equivalencies of Component-Interaction Automata*. Electr. Notes Theor. Comput. Sci. 182, pp. 39–55.

[11] J. Dormoy, O. Kouchnarenko & A. Lanoix (2011): *Runtime Verification of Temporal Patterns for Dynamic Reconfigurations of Components*. In: FACS 2011, LNCS 7253, Springer, pp. 115–132.

[12] J. Dormoy, O. Kouchnarenko & A. Lanoix (2012): *Using Temporal Logic for Dynamic Reconfigurations of Components*. In: FACS 2010, 7th Int. Ws. on Formal Aspects of Component Software, LNCS 6921, Springer, pp. 200–217.

[13] J. Dormoy, O. Kouchnarenko & A. Lanoix (2012): *When Structural Refinement of Components Keeps Temporal Properties Over Reconfigurations*. In: *18th International Symposium on Formal Methods (FM 2012)*, *LNCS* 7436, Springer-Verlag.

[14] A. V. Fioukov, E.M. Eskenazi, D. K. Hammer & M. R. V. Chaudron (2002): *Evaluation of Static Properties for Component-Based Architectures*. In: *28th EUROMICRO Conference 2002*, IEEE Computer Society, pp. 33–39.

[15] R. J. van Glabbeek (1993): *The Linear Time - Branching Time Spectrum II*. In: *CONCUR '93, 4th International Conference on Concurrency Theory*, *LNCS* 715, Springer, pp. 66–81.

[16] V. Grassi, R. Mirandola, E. Randazzo & A. Sabetta (2007): *KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability*. In: *The Common Component Modeling Example: Comparing Software Component Models*, *LNCS* 5153, Springer, pp. 327–356.

[17] A. G. Hamilton (1978): *Logic for mathematicians*. Cambridge University Press, Cambridge.

[18] M. Hennessy & R. Milner (1985): *Algebraic Laws for Nondeterminism and Concurrency*. *Journal of the ACM* 32(1), pp. 137–161.

[19] Y. Kesten, Z. Manna & A. Pnueli (1994): *Temporal Verification of Simulation and Refinement*. In: *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, *LNCS* 803, Springer, pp. 273–346. Available at `http://dx.doi.org/10.1007/3-540-58043-3_22`.

[20] A. Lanoix, J. Dormoy & O. Kouchnarenko (2011): *Combining Proof and Model-checking to Validate Reconfigurable Architectures*. In: *FESCA 2011*, ENTCS.

[21] M. Léger, Th. Ledoux & Th. Coupaye (2010): *Reliable Dynamic Reconfigurations in a Reflective Component Model*. In: *CBSE 2010*, *LNCS* 6092, pp. 74–92.

[22] M. Leuschel & M. J. Butler (2003): *ProB: A Model Checker for B*. In: *Int. Symp. of Formal Methods Europe FME'03*, *LNCS* 2805, Springer, pp. 855–874. Available at `http://springerlink.metapress.com/openurl.asp?genre=article{&}issn=0302-9743{&}volume=2805{&}spage=855`.

[23] M. Leuschel & D. Plagge (2007): *Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more*. In: *ISoLA'07*, *Revue des Nouvelles Technologies de l'Information* RNTI-SM-1, pp. 73–84.

[24] R. Milner (1980): *A Calculus of Communicating Systems*. *Lecture Notes in Computer Science* 92, Springer Verlag.

[25] R. Milner (1989): *Communication and Concurrency*. Prentice-Hall, Inc.

[26] D. Park (1981): *Concurrency and Automata on Infinite Sequences*. In: *Lecture Notes in Computer Science*, 104, Springer Verlag, pp. 167–183.

[27] H. W. Schmidt, I. Crnkovic, G. T. Heineman & J. A. Stafford, editors (2007): *Component-Based Software Engineering, 10th International Symposium, CBSE 2007, Medford, MA, USA, July 9-11, 2007, Proceedings*. *LNCS* 4608, Springer.

A Architectural Configuration Definition [11]

Definition 5 (Configuration) *A configuration c is a tuple* $\langle Elem, Rel \rangle$ *where*

- $Elem = Components \uplus Interfaces \uplus Parameters \uplus Types$ *is a set of architectural elements, such that*
  - $Components$ *is a non-empty set of the core entities, i.e components;*
  - $Interfaces = ReqInterfaces \uplus ProvInterfaces$ *is a finite set of the (required and provided) interfaces;*
  - $Parameters$ *is a finite set of component parameters;*
  - $Types = ITypes \uplus PTypes$ *is a finite set of the interface types and the parameter data types;*
- $Rel = \begin{cases} Container \uplus ContainerType \uplus Parent \\ \uplus Binding \uplus Delegate \uplus State \uplus Value \end{cases}$

  is a set of architectural relations which link architectural elements, such that
  - $Container : Interfaces \uplus Parameters \to Components$ *is a* total function *giving the component which supplies the considered interface or the component of a considered parameter;*
  - $ContainerType : Interfaces \uplus Parameters \to Types$ *is a* total function *that associates a type with each required/provided interface, or with a parameter;*
  - $Parent \subseteq Components \times Components$ *is a* relation *linking a sub-component to the corresponding composite component*[9]*;*
  - $Binding : ProvInterfaces \to ReqInterfaces$ *is a* partial function *which binds together a provided interface and a required one;*
  - $Delegate : Interfaces \to Interfaces$ *is a* partial function *which expresses delegation links;*
  - $State : Components \to \{started, stopped\}$ *is a* total function *giving the status of instantiated components;*
  - $Contingency : ReqInterfaces \to \{mandatory, optional\}$ *is a* total function *to characterise the required interfaces;*
  - $Value : Parameters \to \bigcup_{ptype \in PType} ptype$ *is a* total function *which gives the current value of each parameter.*

[9]For any $(p,q) \in Parent$, we say that q has a sub-component p, i.e. p is a child of q.

B Architectural Consistency Constraints *CC*

$$\forall c.(c \in Components \Rightarrow (\exists ip.(ip \in ProvInterfaces \wedge Container(ip) = c))) \quad \textbf{(CC.1)}$$

$$\forall c, c' \in Components.(c \neq c' \wedge (c, c') \in Parent \Rightarrow \forall\ p.(p \in Parameters \Rightarrow Container(p) \neq c')) \quad \textbf{(CC.2)}$$

$$\forall c, c' \in Components.((c, c') \in Parent^{+} \Rightarrow c \neq c') \quad \textbf{(CC.3)}$$

$$\begin{array}{l}\forall ip \in ProvInterfaces,\\ \forall ir \in ReqInterfaces\end{array} . \left(Binding(ip) = ir \Rightarrow \begin{array}{l} InterfaceType(ip) = InterfaceType(ir) \\ \wedge Container(ip) \neq Container(ir) \end{array} \right) \quad \textbf{(CC.4)}$$

$$\begin{array}{l}\forall ip \in ProvInterfaces,\\ \forall ir \in ReqInterfaces\end{array} . \left(Binding(ip) = ir \Rightarrow \exists c \in Components. \left(\begin{array}{l} (Container(ip), c) \in Parent \\ \wedge (Container(ir), c) \in Parent \end{array} \right) \right) \quad \textbf{(CC.5)}$$

$$\begin{array}{r}\forall ip \in ProvInterfaces,\\ \forall ir \in ReqInterfaces, \forall id \in Interfaces\end{array} . \left(Binding(ip) = ir \Rightarrow \begin{array}{l} Delegate(ip) \neq id \\ \wedge Delegate(ir) \neq id \end{array} \right) \quad \textbf{(CC.6)}$$

$$\forall i, i' \in Interfaces. \left(Delegate(i) = i' \Rightarrow \begin{array}{l} \forall ip.(ip \in ProvInterfaces \Rightarrow Binding(ip) \neq i) \\ \wedge \forall ir.(ir \in ReqInterfaces \Rightarrow Binding(i) \neq ir) \end{array} \right) \quad \textbf{(CC.7)}$$

$$\forall i, i' \in Interfaces.(Delegate(i) = i' \wedge i \in ProvInterfaces \Rightarrow i' \in ProvInterfaces) \quad \textbf{(CC.8)}$$

$$\forall i, i' \in Interfaces.(Delegate(i) = i' \wedge i \in ReqInterfaces \Rightarrow i' \in ReqInterfaces) \quad \textbf{(CC.9)}$$

$$\forall i, i' \in Interfaces. \left(Delegate(i) = i' \Rightarrow \begin{array}{l} InterfaceType(i) = InterfaceType(i') \\ \wedge\ (Container(i), Container(i')) \in Parent \end{array} \right) \quad \textbf{(CC.10)}$$

$$\forall i, i', i" \in Interfaces. \left(\begin{array}{r} (Delegate(i) = i' \wedge Delegate(i) = i" \Rightarrow i' = i") \\ \wedge (Delegate(i) = i" \wedge Delegate(i') = i" \Rightarrow i = i') \end{array} \right) \quad \textbf{(CC.11)}$$

$$\forall ir \in ReqInterfaces. \left(\begin{array}{l} State(Supplier(ir)) = started \\ \wedge Contingency(ir) = mandatory \end{array} \Rightarrow \exists i \in Interfaces. \left(\begin{array}{l} Binding(i) = ir \\ \vee \\ Delegate(i) = ir \end{array} \right) \right) \textbf{(CC.12)}$$

C Architectural Substitutability Constraints SC_{Subst}

$$Parameters_A \subseteq Parameters_R \ \wedge\ PTypes_A \subseteq PTypes_R \qquad \textbf{(SC.1)}$$

$$\forall p.(p \in Parameters_A \Rightarrow (ParamType_A(p) = ParamType_R(p) \wedge Value_A(p) = Value_R(p))) \qquad \textbf{(SC.2)}$$

$$\forall p. \left(\begin{array}{r} p \in Parameters_R \\ \setminus Parameters_A \end{array} \Rightarrow \begin{array}{r} Container_R(p) \in \\ Components_R \\ \setminus Components_A \end{array} \wedge \forall c_A. \left(\begin{array}{r} c_A \in Components_A \setminus Components_R \Rightarrow \\ Subst(c_A) \neq Container_R(p) \end{array} \right) \right) \qquad \textbf{(SC.3)}$$

$$\forall c.(c \in Components_A \cap Components_R \Rightarrow State_A(c) = State_R(c)) \qquad \textbf{(SC.4)}$$

$$\begin{array}{r} \forall c \in Components_A \cap Components_R, \\ \forall x \in Interfaces_A \uplus Parameters_A \end{array} .(Container_A(x) = c \Rightarrow Container_R(x) = c) \qquad \textbf{(SC.5)}$$

$$\begin{array}{r} \forall c \in Components_A \\ \cap Components_R, \\ \forall c' \in Components_A \end{array} . \left((c,c') \in Parent_A \Rightarrow \left(\begin{array}{l} c' \in Components_R \wedge (c,c') \in Parent_R \\ \vee \\ c' \notin Components_R \wedge \\ \quad \exists c" \in Components_R \setminus Components_A. \\ \quad (Subst(c') = c" \wedge (c,c") \in Parent_R) \end{array} \right) \right) \qquad \textbf{(SC.6)}$$

$$\forall c_A. \left(\begin{array}{r} c_A \in Components_A \\ \setminus Components_R \end{array} \Rightarrow \left(\begin{array}{r} \exists c_R \in Components_R \\ \setminus Components_A \end{array} .(Subst(C_A) = c_R) \right) \right) \qquad \textbf{(SC.7)}$$

$$\begin{array}{r} \forall c_A \in Components_A \setminus Components_R, \forall c_R \in Components_R \setminus Components_A \\ .(Subst(c_A) = c_R \Rightarrow State_A(c_A) = State_R(c_R)) \end{array} \qquad \textbf{(SC.8)}$$

$$\begin{array}{r} \forall c_A \in Components_A \setminus Components_R, \forall c_R \in Components_R \setminus Components_A \\ .(Subst(c_A) = c_R \Rightarrow (\forall i \in Interfaces_A.Container_A(i) = c_A \Rightarrow Container_R(i) = c_R)) \end{array} \qquad \textbf{(SC.9)}$$

$$\begin{array}{r} \forall c_A \in Components_A \\ \setminus Components_R, \\ \forall c_R \in Components_R \\ \setminus Components_A, \\ \forall p \in Parameters_A \end{array} . \left(\begin{array}{r} Subst(c_A) = c_R \wedge \\ Container_A(p) = c_A \\ \Rightarrow \end{array} \left(\begin{array}{l} Container_R(p) = c_R \\ \vee \\ \exists c'_R \in Components_R \setminus Components_A. \\ \quad ((c'_R, c_R) \in Parent^+ \wedge Container_R(p) = c'_R) \end{array} \right) \right) \qquad \textbf{(SC.10)}$$

$$\begin{array}{r} \forall c_A \in Components_A \\ \setminus Components_R, \\ \forall c_R \in Components_R \\ \setminus Components_A, \\ \forall c'_A \in Components_A \end{array} . \left(\begin{array}{r} Subst(c_A) = c_R \wedge \\ (c_A, c'_A) \in Parent_A \\ \Rightarrow \end{array} \left(\begin{array}{l} c'_A \in Components_R \wedge (c_R, c'_A) \in Parent_R \\ \vee \\ c'_A \notin Components_R \wedge \\ \exists c'_R \in Components_R \setminus Components_A. \\ ((c_R, c'_R) \in Parent \wedge Subst(c'_A) = c'_R) \end{array} \right) \right) \qquad \textbf{(SC.11)}$$

$$\begin{array}{r} \forall c_R \in Components_R \\ \setminus Components_A, \\ \forall c_A \in Components_A \\ \setminus Components_R \end{array} . \left(\begin{array}{l} Subst(c_A) \neq c_R \Rightarrow \\ \exists c'_R \in Components_R \setminus Components_A. \\ \quad ((c_R, c'_R) \in Parent_R) \end{array} \right) \qquad \textbf{(SC.12)}$$

$$\begin{array}{r} ITypes_A \subseteq ITypes_R \ \wedge\ Interfaces_A \subseteq Interfaces_R \\ \wedge ProvInterfaces_A \subseteq ProvInterfaces_R \wedge\ ReqInterfaces_A \subseteq ReqInterfaces_R \end{array} \qquad \textbf{(SC.13)}$$

$$\forall i.\,(i \in Interfaces_A \Rightarrow InterfaceType_A(i) = InterfaceType_R(i)) \qquad \textbf{(SC.14)}$$

$$\forall i.\,(i \in ReqInterfaces_A \Rightarrow Contingency_A(i) = Contingency_R(i)) \qquad \textbf{(SC.15)}$$

$$\forall i. \left(\begin{array}{l} i \in ReqInterfaces_R \\ \setminus ReqInterfaces_A \end{array} \Rightarrow \begin{array}{r} Container_R(i) \in \\ Components_R \\ \setminus Components_A \end{array} \wedge \forall c_A . \left(\begin{array}{l} c_A \in Components_A \setminus Components_R \\ \Rightarrow Subst(c_A) \neq Container_R(i) \end{array} \right) \right) \qquad \textbf{(SC.16)}$$

$$\forall i. \left(i \in \begin{array}{l} ProvInterfaces_R \\ \setminus ProvInterfaces_A \end{array} \Rightarrow Container_R(i) \in \begin{array}{l} Components_R \\ \setminus Components_A \end{array} \right) \qquad \textbf{(SC.17)}$$

$$\begin{array}{l} \forall pi \in ProvInterfaces_A, \\ \forall ri \in ReqInterfaces_A \end{array} .\,(Binding_A(pi) = ri \Rightarrow Binding_R(pi) = ri) \qquad \textbf{(SC.18)}$$

$$\forall i, i' \in Interfaces_A .(Delegate_A(i) = i' \Rightarrow Delegate_R(i) = i') \qquad \textbf{(SC.19)}$$

$$\begin{array}{l} \forall pi \in ProvInterfaces_R, \forall ri \in ReqInterfaces_R \\ .\left(\left(\begin{array}{l} Binding_R(pi) = ri \\ \wedge Binding_A(pi) \neq ri \end{array} \right) \Rightarrow \left(\begin{array}{l} pi \in ProvInterfaces_R \setminus ProvInterfaces_A \\ \wedge ri \in ReqInterfaces_R \setminus ReqInterfaces_A \end{array} \right) \right) \end{array} \qquad \textbf{(SC.20)}$$

$$\begin{array}{l} \forall i, i' \in Interfaces_R \\ . \left(\begin{array}{l} Delegate_R(i) = i' \\ \wedge Delegate_A(i) \neq i' \end{array} \Rightarrow \left(\begin{array}{l} i \in Interfaces_R \setminus Interfaces_A \\ \wedge \left(\begin{array}{l} i' \in Interfaces_R \setminus Interfaces_A \\ \vee \\ i' \in Interfaces_A \wedge \\ \quad \exists c_A \in Components_A \setminus Components_R . \\ \quad\quad (Subst(c_A) = Container_R(i')) \end{array} \right) \end{array} \right) \right) \end{array} \qquad \textbf{(SC.21)}$$